Incremental Tree Height Reduction For Code Compaction

Alexandru Nicolau* Roni Potasman**
*Information and Computer Science Department
**Dept. of Electrical and Computer Engineering
University of California, Irvine. Irvine, CA. 92717

March 4, 1990

Abstract

This paper introduces a new Tree Height Reduction (THR) technique for code compaction. THR, which is well known parallelizing method, has two interesting properties: while known compilation techniques can get constant factor of speed-up, THR has speed-up of $O(n/\log n)$. Furthermore, THR is able to compact code which seems, at first, uncompactable (due to data dependencies). The algorithm presented is incremental, local (so in each step,it is checking the the current operation and its predecessor rather than the whole expression tree to see whether compaction is possible) and applicable beyond basic block limits. THR is applied after all other optimization techniques, none of which change the *semantics* of the code, have been applied. THR *is* changing the semantics of the code, thus preserving, of course, the correctness of the intermediate and final values. Also, the reduction is controlled according to the resources available—so in case the compaction is feasible but there are not enough resources—it moves to the next operation. The algorithm produces compacted code suited for any tightly coupled multiprocessors (e.g. Very Long Instruction Word (or VLIW) machines). To our knowledge, it is the first local and incremental THR algorithm working across basic blocks boundaries published so far for code compaction.

1 Introduction

Tree Height Reduction is a well known technique for reducing the height of an expression tree from n to $\log n$. Expression trees can be a simple algebraic

statements (e.g. $\{[a*(b-c)+d]+e\}*f$) or serial three address code like:

```
a:= b - c;
d:= a + e;
f:= d * f;
```

The height of the tree (or the length of the code) is the number of steps needed to compute the whole expression. For example, in the following example we need 5 steps to compute the code:

```
cycle 1:    r1 := b - c;
cycle 2:    r2 := a * r1;
cycle 3:    r3 := r2 + d;
cycle 4:    r4 := r3 + e;
cycle 5:    r5 := r4 * f;
```

But, assuming we have at least 3 adders, 3 multipliers and 1 subtractor we can get:

```
cycle 1:    r1 := b - c ;    t1 := d + e ;    r2 := a * f;
cycle 2:    r2 := a * r1;    t3 :=t6 * f ;    r4 :=t2 * r1;
cycle 3:    r3 := d + r2;    r4 :=r2 + t1;    r5 :=t4 + t3;
```

Which shows reduction of height from 5 to 3 steps.

The importance of THR in code compaction is due to the fact that none of the other existing techniques can get rid of the dependencies between these operations— so, without THR, the code can not be compacted any more. By *changing* the code, THR can reduce the length of the code dramatically. Furthermore, previous published compaction techniques gain at most constant factor of speed-up, whereas THR has a speed-up factor of $O(n/\log n)$. Hence, as long as the dependency chain of operations preserve a certain form and there are enough resources, issuing more operations just increases the speed-up. The form, the chain of operations should maintain, is explained in section 3. THR takes advantage of the associativity and distributivity properties of addition and multiplication (and change of sign for subtraction). The algorithm presented, performs well on code which include these 3 kinds of operations. It can be extended easily to code with logical (AND, OR) operations as well. In performing THR care must be taken not to violate numerical and other properties of the expression. However in a big variety of cases this process can be applied.

As mentioned, the algorithm presented is incremental, local and works across basic block limits.
By incremental we mean the ability to perform THR on the whole graph by repeatedly applying it to different operations and the ability to start the process wherever we choose. Local means that we can test adjacent nodes in the graph rather than the whole program.

One of our main goals was to integrate THR into an existing set of local transformations called Percolation Scheduling (PS) [Ni84]. In this context incremental and local THR has some important advantages: it is less ad hoc than global one, it has more general application, it is easier to implement and it interfaces very well with other *local* code transformations (e.g. PS) and enables better control of resources. Since PS itself is a set of local and incremental transformations, which is proven to be complete for all practical purposes [Ai88], having a local and incremental algorithm for THR enables us to apply it naturally across conditional jumps — a major limitation to previous approaches to the exploitation of THR.

Although we have an implemented THR algorithm pipelined operations— we assume throughout this paper that all operations are one- cycle operations. The extension of the algorithm to pipelined operations is beyond the scope of this paper.

Section 2 presents some typical examples where THR may be very useful, section 3 describes the algorithm itself, section 4 gives its implementation into PS, section 5 shows examples for clarifying the algorithm and in section 6 we discuss some properties of our algorithm and the results derived. Section 7 proves the correctness of the algorithm.

2 Tree Height Reduction Applications

Although [?] claims that applying THR to compilers for multioperation machines "would be quite disappointing" we found a large span of applications of THR for these machines. Obviously, if one considers only basic blocks, the chain of dependencies is not long enough to expose the strength of THR, but by looking at the instruction-level parallelism we are able to go past conditional jumps and have a longer chain of operations which improves the potential parallelism. In this section we'll describe some of the applications.

2.1 Sum Of Vector Elements

In vector elements' sum we compute the sum of the elements of the vector (or array). The sum is computed serially by the code:

```
S := 0;
for i=0 to N do
   S := S + a[i];
end;
```

so, we are adding, in each step, the previous computed sum to the current element and we repeat this step N times. This is a very simple chain of dependencies— which can not be reduced to parallel form without *algorithmic* change in the computation. THR will reduce the computation time from O(N) to O($\log N$).

2.2 Dot Product Of 2 Vectors

If we wish to get the dot product of two vectors a[N] and b[N] we have:

```
S := 0;
for i=0 to N do
   S := S + a[i]*b[i];
end;
```

Again, without changing the *content* of the algorithm we need N steps to get the dot product.

2.3 Recurrences Of Vector Elements

Frequently we have a recursive evaluation of vector's elements like in:

```
for i=0 to N do
   a[i] := K * a[i-1] + c[i];
end;
```

Where K is a constant which is known before the loop starts executing. In such a loop we find a dependency between two iterations which can not be eliminated. Under certain conditions, we can reduce the computation time dramatically.

3 Algorithm Description

3.1 Background

The idea behind tree height reduction is to try to compact code at the expense of the additional computation. In a machine where execution of more than one operation per cycle is possible it is natural to utilize all available (unused) resources in order to increase performance. Hence with THR we try to "fill" those instructions which are not full (or have less operations than may be executed in this cycle).

THR uses three algebraic properties: associativity, commutativity and distributivity.

Let's look at a simple example: Suppose we have the following computation and assume that a0, C1,C2 and C3 are available at the beginning of the computation (we'll see later what are the exact conditions for THR):

a1:= a0 + C1;
a2:= a1 + C2;
a3:= a2 + C3;

and the corresponding tree looks like:

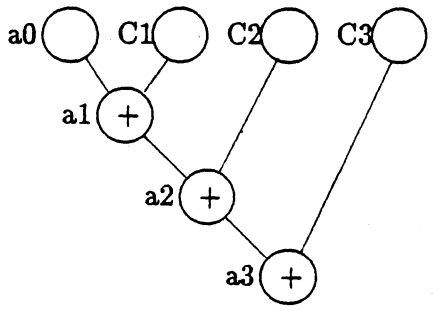

figure 3.1 - original expression tree.

So, if we write the expression for a3 explicitly we get:

a3:= a0 + C1 + C2 + C3;

Using the associativity rule for addition we can write:

a3:= [(a0 + C1) + (C2 + C3)]

And now we can rewrite the computation as:

a1:= a0 + C1; t1:= C2 + C3;

a2:= a1 + C2; a3:= a1 + t1;
Getting tree of height 2 (instead of 3).

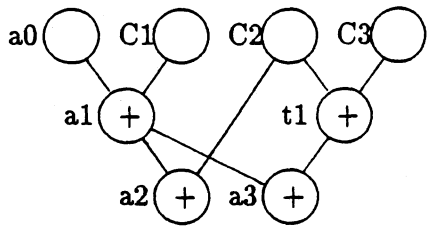

figure 3.2 - compacted expression tree.

The use of commutativity and distributivity for THR will be clarified by the next example. Suppose we have the following computation:

a1:= a0 + C1;
a2:= a1 + C2;
a3:= a2 * C3;
a4:= a3 + C4;
and its tree:

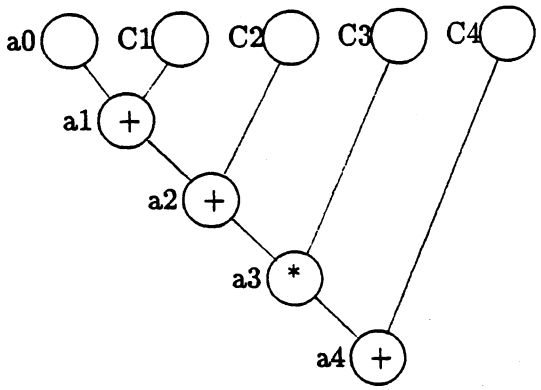

figure 3.3 - original expression tree.

Here we'll use the distributivity of multiplication and transform the the value of a4, which is:

a4:= C4 + C3 * [(a0 + C1) + C2]

into:

a4:= C4 + (C3 * a0) + (C3 * C1) + (C3 * C2)

which can be computed by:

```
t1:= C3 * a0;      t2:= C3 * C1;      t3:= C3 * C2;
t4:= C4 + t1;      t5:= t2 + t3;
a4:= t4 +t5;
```

So, by using three multipliers and two adders, we can reduce the tree height from 4 into 3.

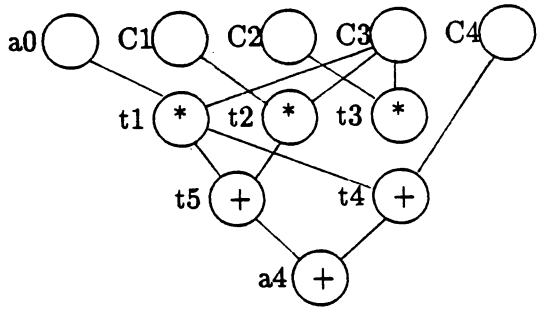

figure 3.4 - compacted expression tree.

3.2 Definitions:

program's structure:
As mentioned, VLIW is one of the best targets of THR. In VLIW machine some operations are compacted into single instruction (or node). Naturally, these operations are performed in parallel— so there shouldn't be data dependency between operations in the same instruction.

operation's structure:

Each operation has an operator (OP_TYPE) and variables which are called *gen* variables (for generation) and *def* variable (for definition). For example in the operation:

a:= b * c;

the *def* is *a* and the *gens* are *b* and *c*. The OP_TYPE is multiplication.

current operation:

The operation currently being examined.

selected_path:

The path in the program selected for THR.

near_son and far_son:

The operations defining the *current operation.* In the example above, *b* and *c* are called the "sons" of *a*. Now, suppose we have the following instructions structure:

instruction (k) :	b:= d + e;	f:= d * g;	
instruction (k+1):	c:= h - e;	i:= d * l;	j:= l + e;
instruction (k+2):	a:= b * c;		

We'll call the operation $(c := h - e)$ *near_son* of operation $(a := b * c)$ while the operation $(b := d + e)$ is called the *far_son* of the *current operation.* The definitions *near* and *far* according to their "distance" from *a*.

available variable

A variable is said to be available in instruction (k) if it is defined at instruction (k-1) or earlier. In the example above *c* is available in instruction (k+2) while *b* is available in instruction (k+1).

Percolate Operations

Percolate operations means scheduling operations as soon as possible, while preserving data dependency and resource constraints. See [EbNi89] for details.

3.3 Algorithm in detail

As mentioned earlier, we apply THR after all other PS transformations have been applied. It means that only those instructions in the graph which are not full, may be considered for THR.

This section describes the algorithm in detail while in the next section we describe its implementation into our PS transformations.

From the examples in previous section it is clear that THR should be performed on paths because the newly added operations, as well as the modified *current operation*, are added to the predecessors of node k. In case of more than one predecessor— a path should be selected.
Although it is usually sufficient to check only adjacent nodes in the program (hence preserve locality)— it turns out that in order to achieve optimality (in the presence of infinite resources) we have to check for the whole chain of operations in the path (or stop wherever the chain is interrupted).
In the following description we'll show how to get the optimal reduction on each path while next section will detail the modified PS transformations needed to keep program integrity.
In our algorithm we distinguish between two cases: the first is due to the use of the associativity property of operations which happens whenever the *current operation* and its *near_son* constitute one of the following pairs: ADD/ADD, ADD/SUB, SUB/ADD, SUB/SUB and MUL/MUL. The other case is when *current operation* is MUL and its *near_son* is either ADD or SUB where the distributivity property is used.

3.3.1 Necessary and sufficient conditions for an operation to be hoisted:

1. One of its *sons* MUST be available at least two instructions earlier than the current one on the path selected.
2. *current operation*'s *near_son* has a *son* which is available at least two instructions earlier than *current operation*'s instruction on that path.
3. If the *current operation* is ADD or SUB the *near_son* has to be either ADD or SUB.(These legal combinations constitute a legal chain).
   If, on the other hand, the *current operation* is MUL
   the *near_son* might be either MUL, ADD or SUB.
4. Both *current operation* and its sons have two gen variables.
5. All relevant instructions on the path have free resources.

3.3.2 Procedures

In this section we'll describe the procedures used in top-down fashion.

```
Procedure THR_Analysis(selected_path)
  FOR all instructions in the selected_path DO
    reset BACK_TRACK flag; /* needed after successful Distributivity_Analysis*/
    FOR all operations in this instruction DO
      IF (the operations meets the necessary conditions) DO
        find which case is it; /* associativity or distributivity */
        SWITCH
          case (associativity) : Associativity_analysis(current operation);
          case (distributivity): Distributivity_Analysis(current operation);
        END /*SWITCH*/
        percolate operations on the path as high as possible;
      END /*IF*/
      IF (one of the operations caused BACK_TRACK)
        check previous_instruction;
      ELSE
        check next_instruction;
    END /*FOR all operations */
  END /*FOR all instructions */
END (Procedure);
```

The algorithm uses distributivity to "push" multiplications toward the path head. The BACK_TRACK flag causes backtracking of the algorithm to the previous instruction. This instruction has to be rechecked due to the possibility of creation of "new" legal chain of operations following the "pushing" of multiplications upward. This idea is shown later explicitly.

```
Procedure Associativity_Analysis(current operation)
  /* preserve correct signs of added operations */
  IF (current operation is SUB and near_son is its second argument)
       set SIGN_FLAG;
  /* find the earliest operation in selected_path satisfying the five conditions */
  earliest_operation=Find_Highest_Avail_Op(current operation);
  IF (succeeded to find such an operation)
       /* add recursively operations to the path */
```

Climb_Up(modified OP_TYPE, earliest_operation's *far_son*,
current_operation's *far_son*);
remove *current operation* from list;
END /*IF*/
END /*Procedure*/

The SIGN_FLAG is responsible for the correct addition of SUB operations into the program. Since $(a-b \neq b-a)$ we need to flip the operands whenever we find an operation whose OP_TYPE is SUB and its *near_son* is its the second argument.

Procedure Distributivity_Analysis(*current operation*)
/* this procedure is called when we have an operation like d:= a*(b+c).
in this case we don't try to hoist d— but rather use the distributivity
property and convert d into d:= a*b + a*c. */
/* add first addant (a*b)*/
add new operation with (MUL_TYPE, *near_son*'s *far_son*,
current operation's i*far_son*) into *near_son*'s instruction;
/* add second addant (a*c)*/
add new operation with (MUL_TYPE, *near_son*'s *near_son*,
current operation's *far_son*) into *near_son*'s instruction;
/* add modified *current operation* (d)*/
add new operation with (*near_son*'s TYPE, first_added_op, second_added_op)
into *current operation*'s instruction;
remove *current operation* from list;
Set the TRACK_BACK flag;
END /*Procedure*/

Procedure Find_Highest_Avail_Op(*selected_path*)
This procedure is searching along the *selected_path* for the earliest
operation which meets the five necessary conditions explained in section 3.4.
The search is recursive by DFS.In order to preserve correctness of the
expression,each time we find an operation which is SUB, meets the
conditions and its *near_son* is the second variable-
we flip the operation sign-so later we call Climb_Up() with modified OP_TYPE.

Procedure Climb_Up(type,first_op,second_op)

```
  /* the procedure is responsible for the addition of new operations into
     the selected_path after we found the earliest operation meeting the
     the conditions by previous procedure. It is called by
     Associativity_Analysis() and by Distributivity_Analysis() first, and
     then calls itself recursively till reaches the near_son of
     current operation. The addition of the modified current operation
     is done by higher level calling procedure (see Move-Op below).*/
  add new operation with (type,first_op,second_op) into the path;
  IF (didn't reach current operation's near_son)
    Climb_Up(first_op's TYPE,first_op's near_son,
      the newly added operation);
END /*Procedure*/
```

4 Implementation Into PS

In this section we present the integration of the algorithm described into PS set of transformations stressing the implementation of THR beyond basic block limits.
The modified Move-Op is an extension of the procedure defined in [EbNi89].

```
Procedure Move-Op(o:operation; n:from-node; m:to-node; p:path-for-move)
  IF (no conflict in m on relevant path) move_possible=TRUE;
  ELSE THR_Analysis(selected_path);
  IF (move_possible) DO
    create a copy n' of n;
    delete all occurrences of o in n'; /* unification */
    move o into tip of path p in m; /* add o as the last operation in tip */
    make m go to n' instead of n (on path p ONLY);
    modify all ops in m writing the same value as o; /*see details [EbNi89]*/
  END /* IF */
  ELSE IF (THR_successful) DO
    create a copy n' of n;
    create a copy m' of m;
    make all predecessors of m (other than the one on the selected_path)
      go to m' instead of m;
    delete all occurrences of o in n'; /* unification */
    move o into tip of path p in m; /* add o as the last operation in tip */
    make m go to n' instead of n (on path p ONLY);
```

```
    modify all ops in m writing the same value as o; /* see details [?] */
    modify o's gen arguments;
    IF (m' has no predecessors) delete(m');
  END /* IF */
  IF (n has no predecessors) delete(n);
END /* Procedure */
```

5 Examples

In this section we present two examples to clarify the algorithm:
example 1:
Suppose that the following code exists for computing elements of a vector:

```
a[1]:=a[0]+C1;
a[2]:=a[1]*C2;
a[3]:=a[2]-C3;
a[4]:=a[3]*C4;
```

Which requires 4 steps to complete.
For the sake of simplicity, assume that a[0] and all the C's are available in the first instruction.
Step 1:
Begin with the third instruction (a[3]:=a[2]-C3). Its *far_son* is NOT defined in the previous instruction, so we enter Associativity_Analysis(). The OP_TYPE is SUB—so we set SIGN_FLAG and proceed to Find_Highest_Avail_Op(). But, because *current operation* is SUB while its *near_son* is MUL (see condition 3) we quit the procedure and advance to next operation.
Step 2:
The *current operation* now is (a[4]:=a[3]*C4). The operation is MUL_TYPE and its *near_son* is SUB so we proceed to Distributivity_Analysis(). Because *near_son*'s type is SUB_TYPE we see that we have to add 3 operations into the tree: the first one is a MUL_TYPE operation with *gen_vars* which are *near_son's far_son* (C3) and *current operation*'s *far_son* (C4). This operation gets a new temporary index (let's assume t1) and is inserted into *near_son*'s instruction. The second operation to be added has the same type as the previous one but its *gen_vars* are (a[2]) and (C4) and its *def_var* is t2. This instruction is inserted into *near_son*'s instruction. The third operation to be added is the reconstruction of *current operation* and it has the TYPE of *near_son* (SUB) and its *gen_vars* are the operations recently added while its

def_var is *current operation*'s *def_var*. After this step the tree has this form:

instruction 1: a[1]:=a[0]+C1;
instruction 2: a[2]:=a[1]*C2;
instruction 3: a[3]:=a[2]-C3; t1:=C3*C4; t2:=a[2]*C4;
instruction 4: a[4]:=a[3]*C4; a[4]:=t1-t2;

After percolation and removal of *current operation* from list we get:

instruction 1: a[1]:=a[0]+C1; t1:=C3*C4;
instruction 2: a[2]:=a[1]*C2;
instruction 3: a[3]:=a[2]-C3; t2:=a[2]*C4;
instruction 4: a[4]:=t2-t1;

Last thing to do in this step in to set the BACK_TRACK flag (which indicates that the next instruction to be examined should be the previous one).

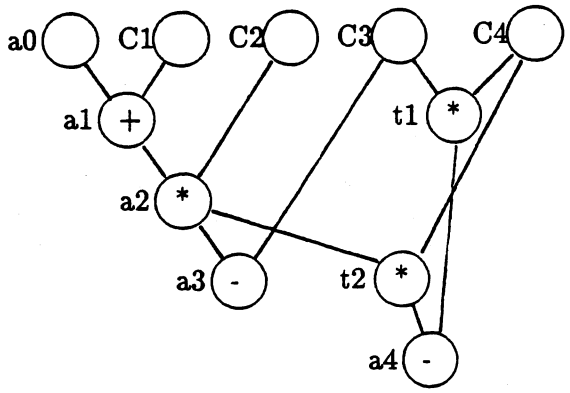

figure 5.1 - example 1 code after step 2.

Step 3:
Because BACK_TRACK flag is set we have to track back to instruction 3. a[3] can't be hoisted for the same reason mentioned in step 1 above- so we proceed to the next operation in this instruction which is (t2:=a[2]*C4). The operation is MUL and its *near_son* (a[2]) is MUL hence we find by Find_Highest_Avail_Op() the highest op which is (a[2]:=a[1]*C2). Now, using Climb_Up() we begin to add operations as follows: the first operation

to be added has *gen_vars* which are highest op's *far_son* (C2) and *current operation*'s *far_son* (C4) and the TYPE is MUL.Let's denote this operation t3. The second operation we add is MUL too, whose *gen_vars* are *near_son*'s *near_son* (a[1]) and the newly added operation (t3). This operation replaces t2. Now, we get the following tree:

```
instruction 1: a[1]:=a[0]+C1;   t1:=C3*C4;
instruction 2: a[2]:=a[1]*C2;   t3:= C2*C4;
instruction 3: a[3]:=a[2]-C3;   t2:=a[1]*t3;      t2:=a[2]*C4;
instruction 4: a[4]:=t2-t1;
```

After percolation and removal we get:

```
instruction 1: a[1]:=a[0]+C1;   t1:=C3*C4;      t3:= C2*C4;
instruction 2: a[2]:=a[1]*C2;   t2:=a[1]*t3;
instruction 3: a[3]:=a[2]-C3;   a[4]:=t2-t1;
```

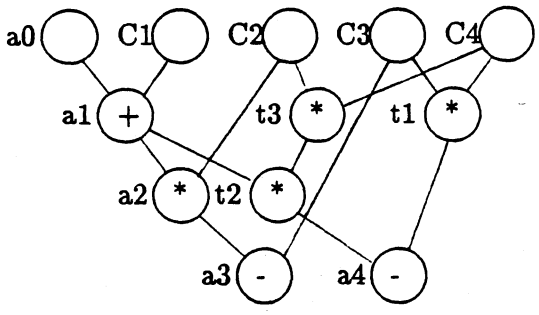

figure 5.2 - example 1 compacted code.

Of course, we got a[4]:= t2-t1 = a[1]*t3 - C3*C4 =
(a[0] + C1)*C2*C4 - C3*C4 = (a[0]+C1-C3)*C4 which is the original value. The procedure shown here reduces the height from 4 to 3 instructions by adding 2 multiplications and one subtraction.

example 2:
Suppose evaluation of elements of a vector of length 15 such as:

a[1] :=a[0] +C1;
a[2] :=a[1] +C2;
a[3] :=a[2] +C3;
a[4] :=a[3] +C4;
a[5] :=a[4] +C5;
a[6] :=a[5] +C6;
a[7] :=a[6] +C7;
a[8] :=a[7] +C8;
a[9] :=a[8] +C9;
a[10]:=a[9] +C10;
a[11]:=a[10]+C11;
a[12]:=a[11]+C12;
a[13]:=a[12]+C13;
a[14]:=a[13]+C14;
a[15]:=a[14]+C15;

which can be executed in 15 steps.

Once again, we assume (only for the sake of simplicity!!) that a[0] and all C's are available at instruction 1. Obviously this computation requires 15 cycles to complete. With the algorithm shown we can get the following tree:

instruction 1:	a[1]:=a[0]+C1;	t1:=C2+C3;	t2:=C4+C5;	t4:=C6+C7;
	t6:=C8+C9;	t8:=C10+C11;	t11:=C12+C13;	t15:=C14+C15;
instruction 2:	a[2]:=a[1]+C2;	a[3]:=a[1]+t1;	t3:=t2+C6;	t5:=t2+t4;
	t7:=t6+C10;	t9:=t6+t8;	t13:=t11+C14;	t16:=t11+t15;
instruction 3:	a[4]:=a[3]+C4;	a[5]:=a[3]+t2;	a[6]:=a[3]+t3;	a[7]:=a[3]+t5;
	t10:=t9+C12;	t12:=t9+t11;	t14:=t9+C13;	t17:=t9+t17;
instruction 4:	a[8]:=a[7]+C8;	a[9]:=a[7]+t6;	a[10]:=a[7]+t7;	a[11]:=a[7]+t9;
	a[12]:=a[7]+t10;	a[13]:=a[7]+t12;	a[14]:=a[7]+t14;	a[15]:=a[7]+t17;

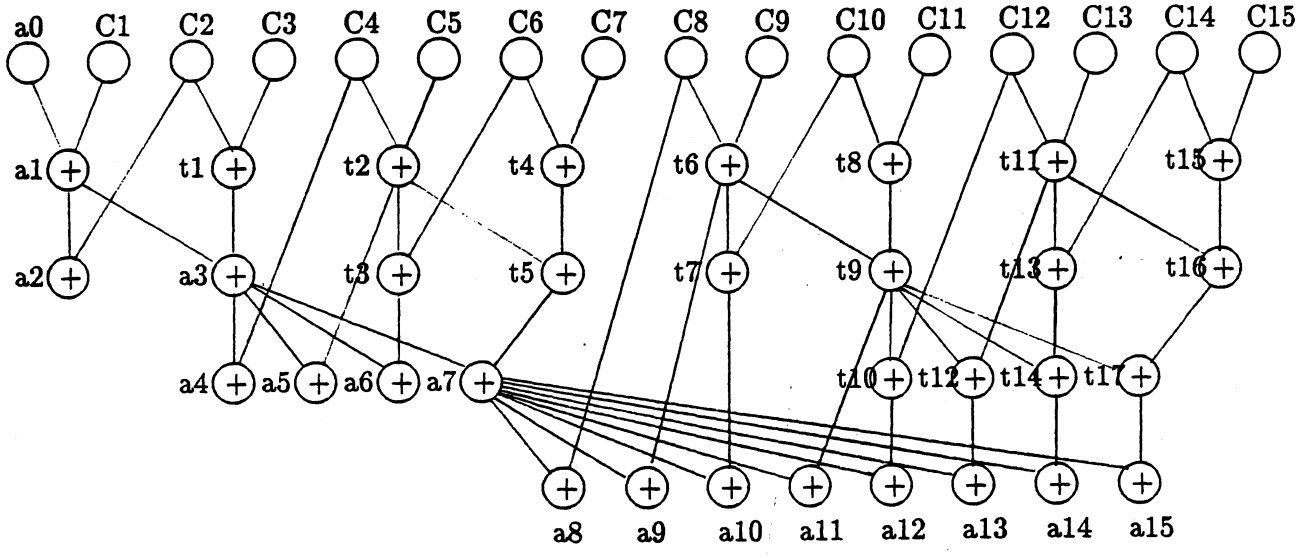

figure 5.3 - example 2 code after compaction.

Here we got a reduction of tree from 15 cycles into 4. The "cost" of the reduction is the addition of 7 adders to each cycle.

example 3:

This example shows how THR works with PS.
Suppose you have the following code:

```
        IF k > l GOTO L1;
        R1 := R0 + C0;
        R2 := R1 + C1;
        R3 := R2 + C2;
        R4 := R3 + C3;
        GOTO L2;
        X1 := X0 + K0;
        X2 := X1 + K1;
L2:     R5 := R4 + X2;
        END;
```

We see that there are two paths, one which has a length of 6 nodes and the other has length of 4. By applying THR on both paths we get the graph shown in figure 5.4 which shows two compacted paths: one of 3 nodes and the other of 2 nodes.

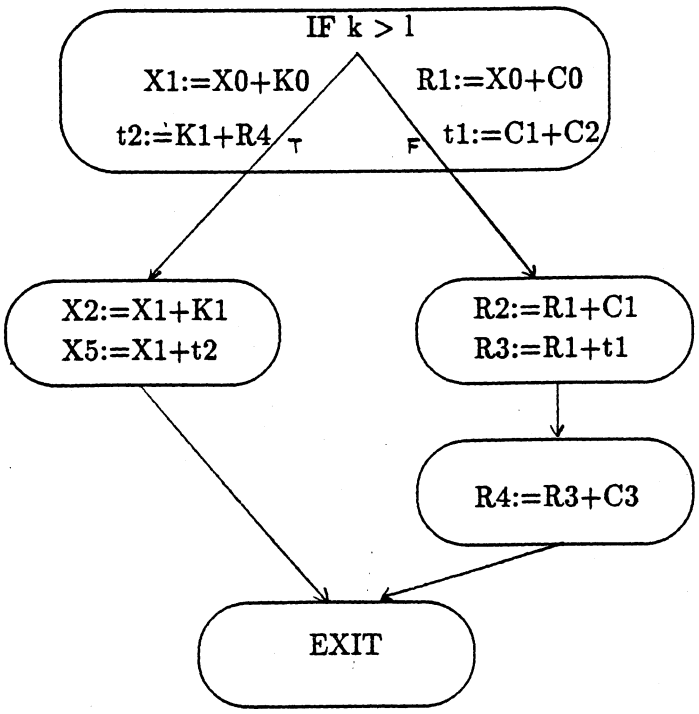

figure 5.4 - example 3 code after compaction.

6 Discussion

Although THR is a well known algorithm - its use in optimizing compilers hasn't been considered seriously for three reasons: a. Researchers were looking mainly at simple algebraic expression trees rather than on any code which has a chain of dependencies. b. In order to use THR efficiently a global view of the code was needed (which was not available, of course). c. THR couldn't be applied across basic blocks, hence it had less impact on code compaction.
But, with the advance in optimizing compilers and especially those with local transformations (like Percolation Scheduling) it is shown that there is a real possibility to compact serial code beyond basic block limits *even with local* analysis of the code.

The speed-up factor of THR, which is $O(n/\log n)$, implies better performance as n grows . It means that by compacting appropriate code we can get a dramatic performance *which can not be obtained by any existing technique.* It means also that as far as we have enough resources- adding more iterations of a loop to the machine improves its speed-up.

Regarding the cost of incremental THR: a successful THR step of a node which is in height **h** from the path header adds between 1 and $\log h - 2$ new operations into the code, while in each node (instruction) it adds **maximum** one new operation. For the optimal case, a code which has a chain of **h** operations whose one operand is available at the code header (e.g. example 2), the compaction to height $\log h$ involves **total** addition of $h/2 - 1$ operations in each node.
We have to emphasize here that sometimes there are intermediate values in the chain which are not needed after the algorithm has been applied. In this case, by dead code elimination techniques, we can reduce the overall number of operations.

7 Proof Of Correctness

In this section we give some theorems and their proofs, to show the correctness of the algorithm.
Theorem 1:

A tree of operations which have chain dependency and length n can't be reduced to height less than $\log n$.

Proof:

A chain dependency of n operations means that the expression to be computed has n+1 variables. Because each operation can be performed on 2 operations only- in the first cycle we can perform not more than (n/2) operations getting (n/2) new variables. So, in each cycle we can reduce the number of variables by 2. that means that the total number of cycles needed is $\log n$.

Theorem 2:

An operation, in order to be hoisted, has to meet the necessary and sufficient conditions of 3.3.

Proof:

As before, it has to be clear that we assume the percolation of all operations in the tree has been done *before* we try to hoist any operation.

Necessity:

If condition 1 is not met it means that both sons are defined in cycle n-1 (where we assume that *current operation*'s cycle is n)- so, it's obvious that this operation can NOT move up because of data dependency conflicts.

If condition 2 is not met it means that *near_son*'s sons are both defined at cycle n-2. It means that the result of these two variables may be obtained only in cycle n-1 (as it is before the hoist) which means that the final evaluation of *current operation*'s value can't be maid *before* cycle n.

If the *current operation* is ADD or SUB and its *near_son* is MUL we have an expression like: a := b + c * (sub_expression). In this case we can use neither associativity nor distributivity to make this expression simpler.

The necessity of conditions 4 and 5 is obvious. Sufficiency:

We'll show that when these five conditions are met we can hoist the operation: Because *current operation* has one variable which is NOT defined in cycle n-2 (from condition 1) and *near_son* has one variable which is NOT defined in cycle n-2 too (condition 2)- we can issue an operation in cycle n-2 which is the result of these two variables (and has the same TYPE as *current operation*) and in cycle n-1 we issue an operation which is the result of the newly added operation and *near_son's near_son.* According to condition 5 there are enough resources for this addition. By associativity and commutativity the value in cycle n-1 is exactly the same as original *current operation.*

Theorem 3:

The algorithm converges.

Proof:
The convergence of the algorithm is not obvious (although we have finite number of operations in the program) because of the BACK_TRACK step we have when we encounter a situation where *current operation* is MUL_TYPE and it's *near_son* is either ADD or SUB. But, because there is no symmetricity between MULs on one hand and ADDs and SUBs on the other hand (i.e. when *current operation* is either ADD or SUB and *near_son* is MUL), hence, all the MUL operations are "pushed" up while the ADDs and the SUBs are "pushed" down. Due to this and the fact that the number of instructions remains unchanged while using distributivity (when *current operation* is MUL while its son isn't)- the convergence is guarantied.

References

[Ai88] A. S. Aiken. Compaction-Based Parallelization. PhD thesis, Cornell University, August 1988.

[AlKe82] J. R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. Technical Report MASC TR 82-6, Rice University, 1982.

[EbNi89] K.Ebcioglu, and A.Nicolau. A *global* resource-constrained parallelization technique. In Proc. ACM SIGARCH ICS-89: International Conference on Supercomputing, Crete, Greece June 2-9 1989.

[Ni84] A. Nicolau. Percolation Scheduling: A parallel compilation technique. Technical Report 85-678, Cornell University, 1984.

[KuMuCh72] D. J. Kuck, Y. Muraoka and S. C. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. IEEE Transactions on Computers, C-21, 12, December 1972.

[TjFl70] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of independent instructions. IEEE Transactions on Computers, Vol. 19, No. 10, October 1970.